Quasi-linear algorithm for the component tree

Laurent Najman, Michel Couprie

Quasi-linear algorithm for the component tree

L. Najman and M. Couprie
Laboratoire A2SI, Groupe ESIEE Cité Descartes, BP99
93162 Noisy-le-Grand Cedex France
{l.najman,m.couprie}@esiee.fr

ABSTRACT

The level sets of a map are the sets of points with level above a given threshold. The connected components of the level sets, thanks to the inclusion relation, can be organized in a tree structure, that is called the *component tree*. This tree, under several variations, has been used in numerous applications. Various algorithms have been proposed in the literature for computing the component tree. The fastest ones have been proved to run in $0(n \ln(n))$ complexity. In this paper, we propose a simple to implement quasi-linear algorithm for computing the component tree on symmetric graphs, based on Tarjan's union-find principle.

Keywords: Component tree, mathematical morphology

1. INTRODUCTION

The level sets of a map are the sets of points with level above a given threshold. The connected components of the level sets, thanks to the inclusion relation, can be organized in a tree structure, that is called the *component tree*. The component tree captures some essential features of the map. Thus, it has been used (under several variations) in numerous applications among which we can cite: image filtering and segmentation,[1–4] video segmentation,[5] image registration,[6,7] image compression[5] and data visualization.[8] We also note that this tree is fundamental for the efficient computation of the topological watershed introduced by M. Couprie and G. Bertrand.[3,9]

While having been (re)discovered by several authors for image processing applications, the component tree concept has its root in statistics.[10,11] For image processing, the use of this tree in order to represent the "meaningful" information contained in a numerical function can be found in particular, in a paper by Hanusse and Guillataud[1,2]; the authors claim that this tree can play a central role in image segmentation, and suggest a way to compute it, based on an immersion simulation. Several authors, such as Vachier,[12] Breen and Jones,[13] Salembier et al.[5] have used this structure in order to implement efficiently some morphological operators (e.g. connected operators, granulometries, extinction functions).

Let us describe informally an "emergence" process that will later help us designing an algorithm for the component tree. We are going to use topographical references. We can see the map as the surface of a relief with the level of a point corresponding to its altitude. Imagine the surface completely covered by water, and that the level of water slowly decreases. Islands (maxima) appear. These islands form the leafs of the component tree. As the level of water decreases, islands will grow, building the branches of the tree. Sometimes, at a given level, several islands will merge into one connected piece. Such pieces are the forks of the tree. We stop when all the water has disappeared.

Various algorithms have been proposed in the literature for computing the component tree,[5,13,14] the last reference also contains a discussion about time complexity of the different algorithms. The fastest ones run in $0(n \ln(n))$ complexity. In this paper, we propose a quasi-linear algorithm for computing the component tree on general symmetric graphs, based on Tarjan's union-find[15] principle. We would like to emphasize that this algorithm is simple to implement. A proof of the complexity of this algorithm is given.

Corresponding author: L. Najman

2. GRAPH STRUCTURE AND THE COMPONENT TREE

2.1. Basic notions for graphs

Let E be a finite set (set of vertices, or points). A *graph* (E, Γ) is defined through a mapping Γ from E to $\mathcal{P}(E)$, where $\mathcal{P}(E)$ denotes the set of all subsets of E. The mapping Γ associates to each point x of E, the set $\Gamma(x)$ of points which are *adjacent to* x. A graph (E, Γ) is *symmetric* if for all x, y of E, $y \in \Gamma(x)$ implies $x \in \Gamma(y)$. If $y \in \Gamma(x)$, the set $\{x, y\}$ is called an *edge* of the graph, and y is called a *neighbor* of x.

Let (E, Γ) be a symmetric graph. Let $X \subseteq E$, and let $x_0, x_n \in X$. A *path* from x_0 to x_n in X is an ordered family $(x_0, x_1, \ldots, x_n)$ of points of X such that $x_{i+1} \in \Gamma(x_i)$, with $i = 0 \ldots n-1$. Let $x, y \in X$, we say that x *is X-connected to* y if there exists a path from x to y in X. The relation "is X-connected to" is an equivalence relation. A *connected component* of X is an equivalence class for the relation "is X-connected to". We say that the a subset $X \subseteq E$ is *connected* if X is made of a single connected component. We say that the graph (E, Γ) is connected if E is connected.

In the sequel of the paper, (E, Γ) denotes a connected symmetric graph, such that for any $x \in E$, x does not belong to $\Gamma(x)$.

2.2. Basic notions for weighted graphs

We denote by $\mathcal{F}(E)$ the set composed of all maps from E to D, where D can be the set of rational numbers or the set of integers. For a map $F \in \mathcal{F}(E)$, the triplet (E, Γ, F) is called a *weighted graph*. For a point $p \in E$, $F(p)$ is called the *weight* or *level* of p. When no ambiguity may occur, we will denote by F the weighted graph (E, Γ, F).

Let $F \in \mathcal{F}(E)$, we define $F_k = \{x \in E; F(x) \geq k\}$ with $k \in \mathbb{Z}$; F_k is called a *cross-section* of F.
A connected component of a section F_k is called a *(level k) component* of F.
A level k component of F that does not contain a level $(k+1)$ component of F is called a *(regional) maximum* of F.
We define $k_{\min} = \min\{F(x), x \in E\}$ and $k_{\max} = \max\{F(x), x \in E\}$, which represent respectively, the minimum and the maximum level in the map F.

While the notions we are dealing with in this paper are defined for general graphs, we are going to illustrate our work with the case of 2D images that we model by weighted graphs. Let $\mathbb{Z}$ denote the set of integers. We choose for E a subset of $\mathbb{Z}^2$. A point $x \in E$ is defined by its two coordinates (x_1, x_2). We choose for Γ the 4-connected adjacency relation defined by, for all $x \in E$, $\Gamma_4(x) = \{y \in E \setminus \{x\};\ |x_1 - y_1| + |x_2 - y_2| \leq 1\}$. We assume that E is connected.

Fig. 1 shows a weighted graph F and four cross-sections of F, between the level $k_{\min} = 1$ and the level $k_{\max} = 4$. The set F_4 is made of two connected components which are regional maxima of F.

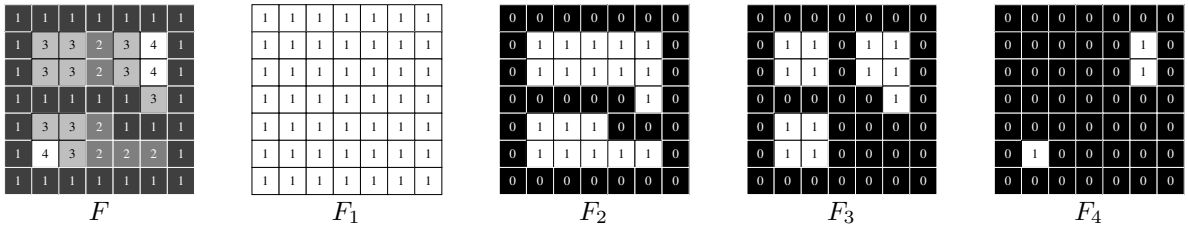


Figure 1. A weighted graph F and its cross-sections at levels 1, 2, 3, 4

2.3. Component Tree

From the example of Fig. 1, we can see that the connected components of the different cross-sections may be organized, thanks to the inclusion relation, to form a tree structure.

Let $F \in \mathcal{F}(E)$, let $k \in D$, we denote by $\mathcal{C}_k(F)$ the set of all connected components of the cross-section F_k, and we define $\mathcal{C}(F)$ as the union of all the sets $\mathcal{C}_k(F)$ with $k \in \{k_{\min} \dots k_{\max}\}$. The elements of $\mathcal{C}(F)$ are the connected components of the cross-sections of F, we will call them *components*.

For the sake of simplicity, we suppose that two components belonging to different cross-sections are always different sets of points, that is, are never strictly identicals. The extension of the following definitions to the general case is straightforward.

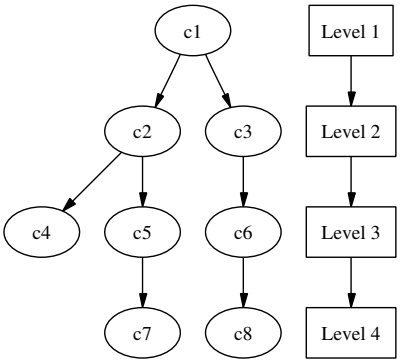
We define the *component tree* $\mathcal{T}(F)$ as the rooted tree such that:

- the *nodes* of the component tree are the elements of $\mathcal{C}(F)$,
- there is an arc from a component $c' \in \mathcal{C}_k(F)$ to a component $c \in \mathcal{C}_j(F)$ if $j = k + 1$ and $c \subseteq c'$. We then say that c' is the *father* of c, and we also say that c is a *son* of c'.

The components which have no sons are called *leafs*, the component which has no father is called the *root*.

We define the *component mapping* C as the map from E to $\mathcal{C}(F)$ which associates to each point $p \in E$ the node $C(p)$ such that $C(p)$ belongs to $\mathcal{C}_k(F)$ with $k = F(p)$, and such that $p \in C(p)$.

Fig. 2.a shows the component tree of the weighted graph depicted in Fig. 1, and fig. 2.b shows the associated component mapping. The component at level 1 is called c_1, the two components at level 2 are called c_2 and c_3 (according to the usual scanning order), and so on.



(a)

c1	c1	c1	c1	c1	c1	c1
c1	c4	c4	c2	c5	c7	c1
c1	c4	c4	c2	c5	c7	c1
c1	c1	c1	c1	c1	c5	c1
c1	c6	c6	c3	c1	c1	c1
c1	c8	c6	c3	c3	c3	c1
c1	c1	c1	c1	c1	c1	c1

(b)

Figure 2. The component tree (a) of the weighted graph of Fig. 1 and the associated component mapping (b)

3. UNION-FIND ALGORITHM FOR THE COMPONENT TREE

3.1. Disjoint Sets

The disjoint set problem consists in maintaining a collection $\mathcal{Q}$ of disjoint subsets of a set E under the operation of union. Each set X in $\mathcal{Q}$ is represented by a unique element of X, called the *canonical element*. In the following, x and y denote two distinct elements of E. Three operations allow to manage the collection:

- **MakeSet(x)**: add the set $\{x\}$ to the collection $\mathcal{Q}$, provided that the element x does not already belongs to a set in $\mathcal{Q}$.
- **Find**(x): return the canonical element of the set in $\mathcal{Q}$ which contains x.
- **Link**(x, y): let X and Y be the two sets in $\mathcal{Q}$ whose canonical elements are x and y respectively (x and y must be different). Both sets are removed from $\mathcal{Q}$, their union $Z = X \cup Y$ is added to $\mathcal{Q}$ and a canonical element for Z is selected and returned.

Tarjan[15] proposed a very simple and very efficient algorithm called *union-find* to achieve any intermixed sequence of such operations with a quasi-linear complexity. More precisely, if m denotes the number of operations and n denotes the number of elements, the worst-case complexity is $\Theta(m\alpha(m,n))$ where $\alpha(m,n)$ is a function which grows very slowly, for all practical purposes $\alpha(m,n)$ is never greater than four.

The implementation of this algorithm is given below in procedure **MakeSet** and functions **Link** and **Find**. Each set of the collection is represented by a rooted tree, where the canonical element of the set is the root of the tree. To each element x is associated a father $\mathrm{Fth}(x)$ and a rank $\mathrm{Rnk}(x)$. The mappings 'Fth' and 'Rnk' are represented by global arrays in memory. One of the two key heuristics to reduce the complexity is a technique called *path compression*, that was used by Tarjan to reduce the cost of **Find**. It consists, while searching for the root r of the tree which contains of x, in considering each element y of the path from x to r (including x), and setting the parent of y to be r. The other key technique, called *union by rank*, consists in always choosing the root with the greatest rank to be the root representative of the union while performing the **Link** operation. If the two roots x and y have the same rank, then one of the roots, say y, is chosen arbitrarily to be the root of the union: y becomes the father x; and the rank of y is incremented by one. The rank $\mathrm{Rnk}(x)$ is a measure of the depth of the tree rooted in x, and is exactly the depth of this tree if the path compression technique is not used jointly with the union by rank technique. Union by rank avoids creating degenerate trees, and helps keeping the depth of the trees as small as possible. For a more detailed explanation and complexity analysis, see Tarjan's paper.[15]

Procedure `MakeSet(`*element* x`)`

$\mathrm{Fth}(x) := x$; $\mathrm{Rnk}(x) := 0$;

Function `element Find (`*element* x`)`

if *(Fth*$(x) \neq x$*)* **then** $\mathrm{Fth}(x) := \mathrm{Find}(\mathrm{Fth}(x))$;
return $\mathrm{Fth}(x)$;

Function `element Link (`*element* x*, element* y`)`

if *(Rnk*$(x) >$ *Rnk*(y)*)* **then** exchange(x, y);
if *(Rnk*$(x) ==$ *Rnk*(y)*)* **then** $\mathrm{Rnk}(y) := \mathrm{Rnk}(y) + 1$;
$\mathrm{Fth}(x) := y$;
return y;

3.2. Illustration of union-find: building the connected components

We can illustrate the use of the union-find algorithm on the classical problem of finding the connected components of a subset X of a graph (E, Γ). The algorithm 1 (BuildConnectedComponents) is given below. For a set X, this algorithm returns a map M that gives for each point p, the canonical element $M(p)$ of the connected component of X which contains p.

A first pass (loop 1) is made, for the union-find preprocessing; during this first pass, for each point p of the set X, the set $\{p\}$ is added to the collection $\mathcal{Q}$ of disjoint subsets. Then, a scanning (loop 2) processes all points of X in any order. For each point p, we first find the canonical element of the disjoint set it belongs to (line 3). Then, for each neighbor q of p such that $q \in X$ (line 4), we find the canonical element of the disjoint set which contains q (line 5). If p and q are not already in the same disjoint set, that is if the two canonical elements differ (line 6), then p and q are linked (line 7), and one of the two canonical elements is chosen to be the canonical element of their common disjoint set. At the end of the scanning, a simple pass on all the elements of X (loop 8) builds the map M.

Algorithm 1: BuildConnectedComponents

Data: (E, Γ) - graph
Data: A set $X \subseteq E$
Result: M - map from X to E

```
1 foreach p ∈ X do MakeSet(p);
2 foreach p ∈ X do
3     compp := Find(p);
4     foreach q ∈ Γ(p) such that q ∈ X do
5         compq := Find(q);
6         if (compp ≠ compq) then compp := Link(compq, compp);
8 for p ∈ X do M(p) := Find(p);
```

3.3. Quasi-linear component tree building algorithm

We are now ready to introduce our quasi-linear algorithm for building the component tree $\mathcal{T}(F)$ from a weighted graph F.

To represent a node of $\mathcal{T}(F)$, we use a structure called `node` containing the level of the node, and the list of nodes which are sons of the current node. For building the component tree, we do not need the reverse link, that is we do not need to know the father of a given node, but let us note that such an information is useful for applications, and can easily be obtained in a linear-time post-processing step.

The algorithm 2 (BuildComponentTree) is given below. We are going to use two collections $\mathcal{Q}_1$ and $\mathcal{Q}_2$ of disjoint sets. After a preprocessing (line 1) for sorting the points by decreasing order of level and for the preparation of the two union-find implementations (line 2), we process the points, starting with the highest ones.

Let us suppose that we have processed a number of levels, and thus that we have already built several parts of the component tree, that is several *partial* trees. Let us examine a given point p of the current level. The first collection $\mathcal{Q}_1$ is used to tell which partial tree the point p belongs to (line 4). Using *subtreeRoot*, we can know the node that is the root of this partial tree; this node may result of the union of differents nodes of the same level. The second collection $\mathcal{Q}_2$ of disjoint sets is used to obtain the canonical element of this union (line 5), that we call *canonical node of p*.

We then look at each neighbor q of level greater or equal to the one of the point p under examination (loop 6). Note that as the graph is symmetric, the **Link** between two points is done when one of the two points is processed as a neighbor of the other. Thus, we can use the order of scanning of the points, and we only need to examine the "already processed" neighbors of p. Such a neighbor q satisfies $F(q) \geq F(p)$.

Exactly as we have done for the point p, we search for the canonical node of q (lines 7-8). If the canonical node of p and the canonical node of q differ, that is if the two points are not already in the same node, we have two possible cases:

- either the two canonical nodes have the same level; in this case, that means that these two nodes are in fact part of the same component, and we have to merge the two nodes (line 9 and function **MergeNodes**). The merging of nodes of same level is done by the second union-find implementation. The merging relies on the fact that the **Link** function always chooses one of the two canonical elements of the sets that are to be merged as the canonical element of the merged set.

  Once the merging has been done, observe that we will not access anymore to the node that has not been chosen to be the canonical element of the set. We will only have to know to which set it belongs to, and the answer to this question is given by the **Find** part of the union-find algorithm.

- either the canonical node of q is of higher level, and thus this node becomes a son of the current node (line 10).

Algorithm 2: BuildComponentTree

Data: (E, Γ, F) - weighted graph with N points.

Result: N_n - number of nodes of the component tree ($\leq N$).

Result: *nodes* - array $[0 \dots N-1]$ of nodes.

Result: *Root* - Root of the component tree

Result: C - map from E to $[0 \dots N-1]$ (component mapping).

Local: *subtreeRoot* - map from $[0 \dots N-1]$ to $[0 \dots N-1]$.

```
1  N_n := N; Sort the points in decreasing order of level for F;
2  foreach p ∈ E do {MakeSet1(p); MakeSet2(p); nodes[p]:= MakeNode(F(p)); subtreeRoot[p] := p;};
3  foreach p ∈ E in decreasing order of level for F do
4      curCanonicalElt := Find1(p);
5      curNode := Find2(subtreeRoot[curCanonicalElt]);
6      foreach already processed neighbor q of p with F(q) ≥ F(p) do
7          adjCanonicalElt := Find1(q);
8          adjNode := Find2(subtreeRoot[adjCanonicalElt]);
           if (curNode ≠ adjNode) then
               if (nodes[curNode]→level == nodes[adjNode]→level) then
9                  curNode := MergeNodes(adjNode, curNode);
               else
                   //We have nodes[curNode]→level < nodes[adjNode]→level
10                 nodes[curNode]→addSon(nodes[adjNode]);
11             curCanonicalElt := Link1(adjCanonicalElt, curCanonicalElt);
12             subtreeRoot[curCanonicalElt] := curNode;
13 Root := subtreeRoot[Find1(Find2(0))] ;
14 foreach p ∈ E do C(p) := Find2(p);
```

Function `node MakeNode(`*int level*`)`

```
allocate a new node n with an empty list of sons;
n→ level := level;
return n;
```

Function `int MergeNodes(`*int node*1, *int node*2`)`

```
tmpNode := Link2(node1,node2);
if (tmpNode == node2) then
    Add the list of sons of nodes[node1] to the list of sons of nodes[node2];
else
    Add the list of sons of nodes[node2] to the list of sons of nodes[node1];
N_n := N_n − 1;
return tmpNode;
```

In both cases, we have to link the two partial trees, this is done using the first Link implementation (line 11). We also have to keep track of the node of lowest level for the union of the two partial trees, that we store in the array *subtreeRoot* (line 12).

At the end of the algorithm, we have to do a post processing to return the desired result. The root of the component tree can easily be found (line 13) using the array *subtreeRoot* and the two disjoint sets $\mathcal{Q}_1$ and $\mathcal{Q}_2$. The component mapping can be obtained using the second disjoint set $\mathcal{Q}_2$ (loop 14).

3.4. Complexity analysis

Let n denotes the number of points in E, and let m denotes the number of edges of the graph (E, Γ).

The sorting of the points (line 1) can be done in linear time if the weigths are small integers (counting sort[16]), and in $n \log(\log(n))$ if the length of each weight can be stored in a machine memory word (long integers or floating point numbers[17]).

Loop 2 is the preparation for the union-find algorithm. It is obviously linear.

In the function **MergeNodes**, the merging of the lists of sons can be done in constant time, because we can merge two lists by setting the first member of one list to be the one that follows the last member of the other list. This requires the two lists to be disjoint, which is the case (we are dealing with disjoint sets), and an adequate representation for lists (chained structure with pointers both on first and last element).

The amortized complexity of line 6 is equal to the number m of edges of the graph (E, Γ). The amortized complexity of all calls to the union-find procedures is quasi-linear with respect to m. The building of the component mapping C is obviously linear.

Thus the complexity of the algorithm 2 (BuildComponentTree) is quasi-linear if the sorting step is linear.

4. EXAMPLE

We are going to illustrate the work of the algorithm on an example. Let us look at the weighted graph of fig. 3.a. The points are numbered (labeled) according to their usual lexicographical order (fig. 3.b). They are examined by the algorithm according to their level, starting with the highest ones.

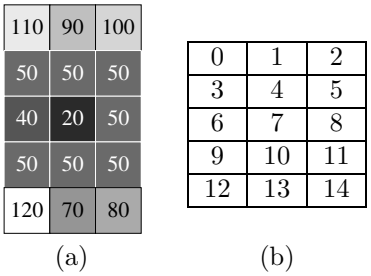


Figure 3. (a) original weighted graph - (b) Points are numbered (label) according to the usual lexicographic order, but they will be processed by decreasing grey level (that is: $12, 0, 2, 1, 14, 13, 3, 4, 5, 8, 9, 10, 11, 6, 7$).

At the begining of the sixth step (end of fifth step), we have already constructed parts of the component tree (fig. 4.b). We have represented the maps Fth1, Fth2, and *subtreeRoot* (fig. 4.a). For the maps Fth1 and Fth2, the canonical elements appear in white. It should be noted that the *subtreeRoot* mapping is only used for the canonical elements of Fth1: this explains why the values of subtreeRoot for other elements (in grey) are not updated.

We are going to process points at level 50. The first point at level 50 is the number 3. The point 0 is a neighbor of 3. The node 0 belongs to the partial tree whose root is the node 1 at level 90. Thus the node 3 becomes the father of node 1. Then, the node 3 is linked succesively with nodes numbered 4, 5 and 8. Then point number 9 is examined, and its node is linked with node number 10, the node 9 being chosen as the

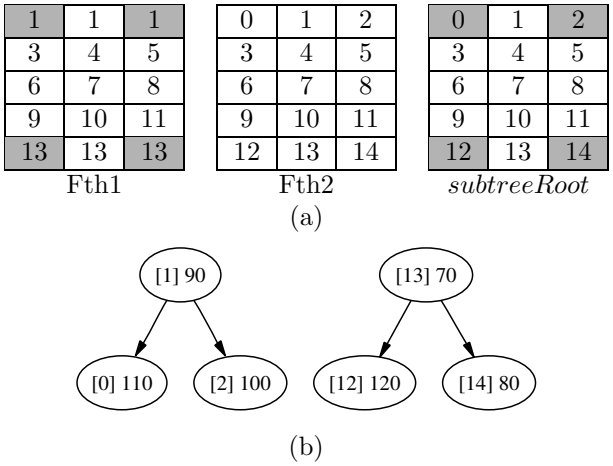


Figure 4. Begining of step 6. (a) State of the map Fth1, Fth2 and *subtreeRoot*. (b) Partial trees already constructed

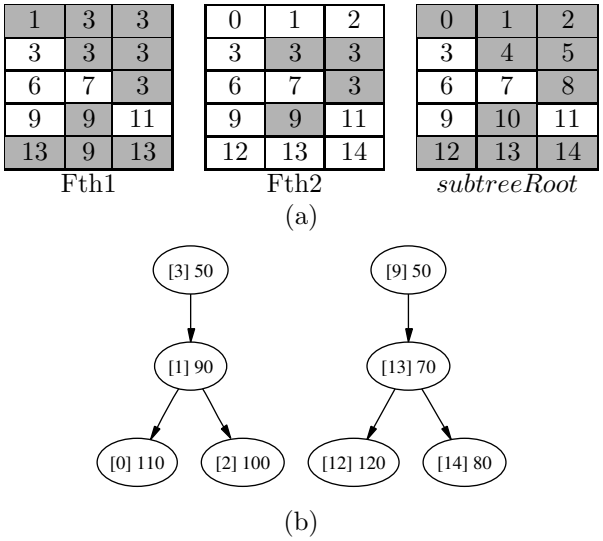


Figure 5. Begining of step 11. (a) State of the map Fth1, Fth2 and *subtreeRoot*. (b) Partial trees already constructed

canonical element. Then, through the point 12, the partial tree whose root is node 13 (level 70) is put under the node 9. We are then at the begining of step 11, and this is illustrated on figure 5.

The point 11 is a neighbor of both points 8 and 10. The node 8 is on the level 50 component whose root is 3. Thus, nodes 11 and nodes 3 are linked, and the node 3 is chosen as the canonical element. The node 10 is on the level 50 component whose root is 9. Thus, nodes 9 and 3 are merged, that is the corresponding partial trees become a single partial tree. The node 9 is chosen as the canonical element of the level 50 component, and the sons of node 3 are transfered to node 9. We are in the situation depicted in figure 6.

We then process the point 6 at level 40, whose node becomes the father of node 9 at level 50. But the union-find algorithm chooses 9 as the canonical element for the partial tree whose root is 6. We thus have to keep the root of the partial tree in *subtreeRoot* by setting $subtreeRoot[9] := 6$. Then we process the point 7 at level 20, and the node 7 becomes the father of node 6. Once again, the union-find algorithm choose 9 as the canoncial element, and thus we have to keep the root of the partial tree by setting $subtreeRoot[9] := 7$. There is no point higher than 20, and thus, the component tree is built. The final situation is depicted in figure 7.

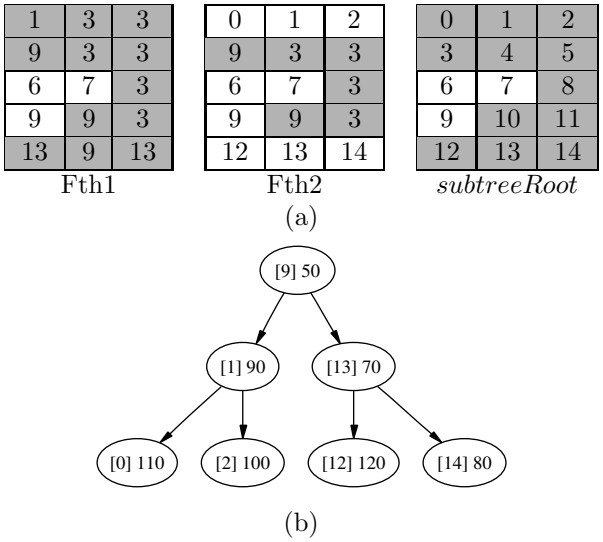


Figure 6. End of step 11. (a) State of the map Fth1, Fth2 and *subtreeRoot*. (b) Partial trees already constructed

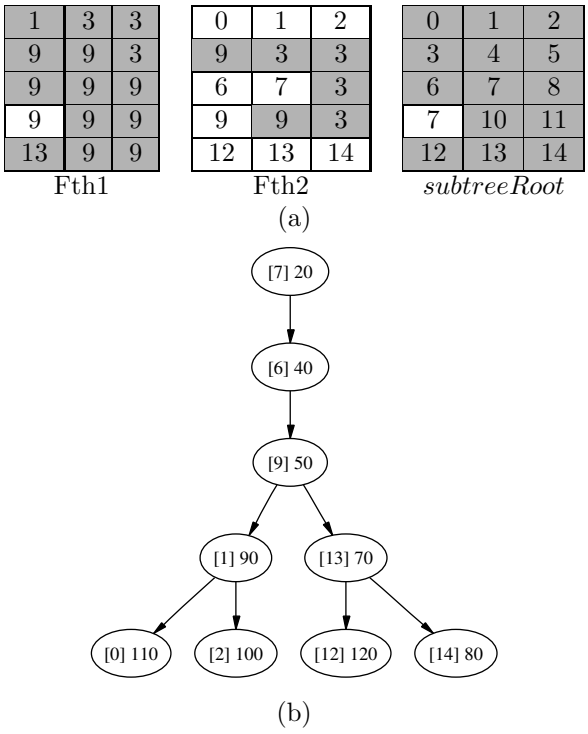


Figure 7. End of step 14. (a) State of the map Fth1, Fth2 and *subtreeRoot*. (b) Component tree

The first collection $\mathcal{Q}_1$ of disjoint sets is not useful anymore: indeed, each node of the graph has been examined, and they are all linked, the canonical element being the node 9. The root of the component tree is the node 7. The second collection $\mathcal{Q}_2$ gives a canonical element for each of the components of F: observe in particular the level 50, whose canonical element is the node 9. The collection $\mathcal{Q}_2$ can be used to compute the component mapping C.

REFERENCES

1. P. Hanusse and P. Guillataud, "Sémantique des images par analyse dendronique," in *8th Conf. Reconnaissance des Formes et Intelligence Artificielle*, **2**, pp. 577–588, AFCET, (Lyon), 1992.
2. P. Guillataud, *Contribution à l'analyse dendroniques des images.* PhD thesis, Université de Bordeaux I, 1992.
3. M. Couprie and G. Bertrand, "Topological grayscale watershed transform," in *SPIE Vision Geometry V Proceedings*, **3168**, pp. 136–146, 1997.
4. R. Jones, "Component trees for image filtering and segmentation," in *NSIP'97*, 1997.
5. P. Salembier, A. Oliveras, and L. Garrido, "Anti-extensive connected operators for image and sequence processing," *IEEE Trans. on Image Proc.* **7**, pp. 555–570, April 1998.
6. J. Mattes, M. Richard, and J. Demongeot, "Tree representation for image matching and object recognition," in *DGCI 99*, G. Bertrand, M. Couprie, and L. Perroton, eds., *LNCS*, pp. 298–309, 1999.
7. P. Monasse, *Morphological representation of digital images and application to registration.* PhD thesis, Paris-Dauphine University, June 2000.
8. Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote, "Simple and optimal output-sensitive construction of contour trees using monotone paths," tech. rep., Polytechnic University, Brooklin, NY, USA, 2003. http://cis.poly.edu/chiang/contour.pdf.
9. M. Couprie, L. Najman, and G. Bertrand, "Quasi-linear algorithms for topological watershed," *Journal of Mathematical Imaging and Vision* , 2004. submitted.
10. D. Wishart, "Mode analysis: A generalization of the nearest neighbor which reduces chaining effects," in *Numerical Taxonomy*, A. Cole, ed., pp. 282–319, Academic Press, (London), 1969.
11. J. Hartigan, "Statistical theory in clustering," *J. of Classification* **2**, pp. 63–76, 1985.
12. C. Vachier, *Extraction de caractéristiques, segmentation d'images et Morphologie Mathématique.* PhD thesis, École National Supérieure des Mines de Paris, 1995.
13. E. Breen and R. Jones, "Attribute openings, thinnings and granulometries," *Computer Vision and Image Understanding* **64**, pp. 377–389, November 1996.
14. J. Mattes and J. Demongeot, "Efficient algorithms to implement the confinement tree," in *9th Conf. on D.G.C.I.*, *LNCS* **1953**, pp. 392–405, Springer Verlag, 2000.
15. R. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM* **22**, pp. 215–225, 1975.
16. T. H. Cormen, C. L. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.
17. A. Andersson, T. Hagerup, S. Nilsson, and R. Raman, "Sorting in linear time?," in *STOC: ACM Symposium on Theory of Computing*, 1995.